23 Testing your application

This chapter covers

- Creating unit test projects with xUnit
- Writing unit tests for custom middleware and API controllers
- Using the Test Host package to write integration tests
- Testing your real application's behavior with `WebApplicationFactory`
- Testing code dependent on EF Core with the in-memory database provider

When I first started programming, I didn't understand the benefits of automated testing. It involved writing so much more code—wouldn't it be more productive to be working on new features instead? It was only when my projects started getting bigger that I appreciated the advantages. Instead of having to manually run my app and test each scenario, I could press Play on a suite of tests and have my code tested for me automatically.

Testing is universally accepted as good practice, but how it fits into your development process can often turn into a religious debate. How many tests do you

need? Is anything less than 100% coverage of your code base adequate? Should you write tests before, during, or after the main code?

This chapter won't address any of those questions. Instead, I'll focus on the *mechanics* of testing an ASP.NET Core application. I'll show you how to use isolated *unit tests* to verify the behavior of your services in isolation, how to test your API controllers and custom middleware, and how to create *integration tests* that exercise multiple components of your application at once. Finally, I'll touch on the EF Core in-memory provider, a feature that lets you test components that depend on a `DbContext` without having to connect to a database.

> **TIP** For a broader discussion of testing, or if you're brand new to unit testing, see *The Art of Unit Testing*, 3rd ed., by Roy Osherove (Manning, 2021). If you want to explore unit test best practices using C# examples, see *Unit Testing Principles, Practices, and Patterns* by Vladimir Khorikov (Manning, 2020). Alternatively, for an in-depth look at testing with xUnit in .NET Core, see *.NET Core in Action* by Dustin Metzgar (Manning, 2018).

In section 23.1 I'll introduce the .NET SDK testing framework and how you can use it to create unit testing apps. I'll describe the components involved, including the testing SDK and the testing frameworks themselves, like xUnit and MSTest. Finally, I'll cover some of the terminology I'll use throughout the chapter.

In section 23.2 you'll create your first test project. You'll be testing a simple class at this stage, but it'll allow you to come to grips with the various testing concepts involved. You'll create several tests using the xUnit test framework, make assertions about the behavior of your services, and execute the test project both from Visual Studio and the command line.

In sections 23.3 and 23.4, we'll look at how to test common features of your ASP.NET Core apps: API controllers and custom middleware. I'll show you how to write isolated unit tests for both, much like you would any other service, and I'll point out the tripping points to watch for.

To ensure components work correctly, it's important to test them in isolation. But you also need to test that they work correctly in a middleware pipeline. ASP.NET Core provides a handy Test Host package that lets you easily write these *integration tests* for your components. You can even go one step further with the `WebApplicationFactory` helper class, and test that your *app* is working correctly. In section 23.5 you'll see how to use `WebApplicationFactory` to simulate requests to your application and to verify that it generates the correct response.

In the final section of this chapter, I'll demonstrate how to use the SQLite database provider for EF Core with an in-memory database. You can use this provider to test services that depend on an EF Core `DbContext`, without having to use a real database. That avoids the pain of having unknown database infrastructure, of resetting the database between tests, and of different people having slightly different database configurations.

Let's start by looking at the overall testing landscape for ASP.NET Core, the options available to you, and the components involved.

23.1 An introduction to testing in ASP.NET Core

In this section you'll learn about the basics of testing in ASP.NET Core. You'll learn about the different types of tests you can write, such as unit tests and integration tests, and why you should write both types. Finally, you'll see how testing fits into ASP.NET Core.

If you have experience building apps with the full .NET Framework or mobile apps with Xamarin, then you might have some experience with unit testing frameworks. If you were building apps in Visual Studio, the steps for creating a test project differed between testing frameworks (xUnit, NUnit, MSTest), and running the tests in Visual Studio often required installing a plugin. Similarly, running tests from the command line varied between frameworks.

With the .NET SDK, testing in ASP.NET Core and .NET Core is now a first-class citizen, on a par with building, restoring packages, and running your application. Just as you can run `dotnet build` to build a project, or `dotnet run` to execute it, you can use `dotnet test` to execute the tests in a test project, regardless of the testing framework used.

The `dotnet test` command uses the underlying .NET SDK to execute the tests for a given project. This is exactly the same as when you run your tests using the Visual Studio test runner, so whichever approach you prefer, the results are the same.

Test projects are console apps that contain a number of *tests*. A test is typically a method that evaluates whether a given class in your app behaves as expected. The test project will typically have dependencies on at least three components:

- The .NET Test SDK
- A unit testing framework, such as xUnit, NUnit, Fixie, or MSTest
- A test-runner adapter for your chosen testing framework, so that you can execute your tests by calling `dotnet test`

These dependencies are normal NuGet packages that you can add to a project, but they allow you to hook in to the `dotnet test` command and the Visual Studio test runner. You'll see an example .csproj file from a test app in the next section.

Typically, a test consists of a method that runs a small piece of your app in isolation and checks that it has the desired behavior. If you were testing a `Calculator` class, you might have a test that checks that passing the values `1` and `2` to the `Add()` method returns the expected result, `3`.

You can write lots of small, isolated tests like this for your app's classes to verify that each component is working correctly, independent of any other components. Small isolated tests like these are called *unit tests*.

Using the ASP.NET Core framework, you can build apps that you can easily unit test; you can test some aspects of your controllers in isolation from your action filters and model binding. This is because the framework

- Avoids static types
- Uses interfaces instead of concrete implementations

- Has a highly modular architecture; for example, you can test your controllers in isolation from your action filters and model binding

But just because all your components work correctly independently doesn't mean they'll work when you put them together. For that, you need *integration tests*, which test the interaction between multiple components.

The definition of an integration test is another somewhat contentious issue, but I think of integration tests as any time you're testing multiple components together, or you're testing large vertical slices of your app: testing a user manager class that can save values to a database, for example, or testing that a request made to a health-check endpoint returns the expected response. Integration tests don't necessarily include the *entire* app, but they definitely use more components than unit tests.

> **NOTE** I don't cover UI tests which, for example, interact with a browser to provide true end-to-end automated testing. Selenium (www.seleniumhq.org) and Cypress (www.cypress.io) are two of the best known tools for UI testing.

ASP.NET Core has a couple of tricks up its sleeve when it comes to integration testing. You can use the Test Host package to run an in-process ASP.NET Core server, which you can send requests to and inspect the responses. This saves you from the orchestration headache of trying to spin up a web server on a different process, making sure ports are available, and so on, but still allows you to exercise your whole app.

At the other end of the scale, the EF Core SQLite in-memory database provider lets you isolate your tests from the database. Interacting with and configuring a database is often one of the hardest aspects of automating tests, so this provider lets you sidestep the issue entirely. You'll see how to use it in section 23.6.

The easiest way to get to grips with testing is to give it a try, so in the next section you'll create your first test project and use it to write unit tests for a simple custom service.

23.2 Unit testing with xUnit

In this section you'll learn how to create unit-test projects, how to reference classes in other projects, and how to run tests with Visual Studio or the .NET CLI. You'll create a test project and use it to test the behavior of a basic currency-converter service. You'll write some simple unit tests that check that the service returns the expected results and that it throws exceptions when you expect it to.

As I described in section 23.1, to create a test project you need to use a testing framework. You have many options, such as NUnit or MSTest, but the most commonly used test framework with .NET Core is xUnit (https://xunit.net/). The ASP.NET Core framework project itself uses xUnit as its testing framework, so it's become somewhat of a convention. If you're familiar with a different testing framework, then feel free to use that instead.

23.2.1 Creating your first test project

Visual Studio includes a template to create a .NET Core xUnit test project, as shown in figure 23.1. Choose File > New Project and choose xUnit Test Project (.NET Core) from the New Project dialog box. Alternatively, you could choose Unit Test Project (.NET Core) or NUnit Test Project (.NET Core) if you're more comfortable with those frameworks.

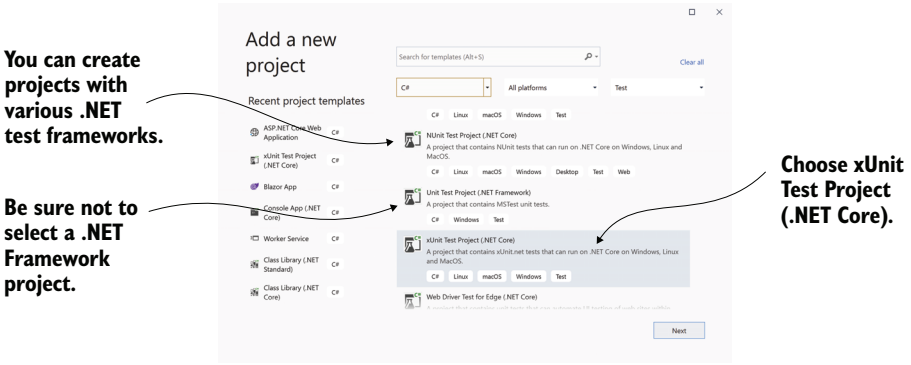


Figure 23.1 The New Project dialog box in Visual Studio. Choose xUnit Test Project to create an xUnit project, or choose Unit Test Project to create an MSTest project.

Alternatively, if you're not using Visual Studio, you can create a similar template using the .NET CLI with

```
dotnet new xunit
```

Whether you use Visual Studio or the .NET CLI, the template creates a console project and adds the required testing NuGet packages to your .csproj file, as shown in the following listing. If you chose to create an MSTest (or other framework) test project, the xUnit and xUnit runner packages would be replaced with packages appropriate to your testing framework of choice.

Listing 23.1 The .csproj file for an xUnit test project

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
        Include="Microsoft.NET.Test.Sdk" Version="16.8.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
```

The test project is a standard .NET Core project targeting .NET 5.0.

The .NET Test SDK, required by all test projects

The xUnit test framework

```
    <PackageReference
        Include="xunit.runner.visualstudio" Version="2.4.3" />
    <PackageReference Include="coverlet.collector" Version="1.3.0" />
  </ItemGroup>
</Project>
```

The xUnit test adapter for the .NET Test SDK

An optional package that collects metrics about how much of your code base is covered by tests

In addition to the NuGet packages, the template includes a single example unit test. This doesn't *do* anything, but it's a valid xUnit test all the same, as shown in the following listing. In xUnit, a test is a method on a public class, decorated with a `[Fact]` attribute.

Listing 23.2 An example xUnit unit test, created by the default template

```
public class UnitTest1
{
    [Fact]
    public void Test1()
    {
    }
}
```

xUnit tests must be in public classes.

The [Fact] attribute indicates the method is a test method.

The Fact must be public and have no parameters.

Even though this test doesn't test anything, it highlights some characteristics of xUnit `[Fact]` tests:

- Tests are denoted by the `[Fact]` attribute.
- The method should be public, with no method arguments.
- The method is `void`. It could also be an `async` method and return `Task`.
- The method resides inside a public, non-static class.

NOTE The `[Fact]` attribute, and these restrictions, are specific to the xUnit testing framework. Other frameworks will use other ways to denote test classes and have different restrictions on the classes and methods themselves.

It's also worth noting that, although I said that test projects are console apps, there's no `Program` class or `static void main` method. Instead, the app looks more like a class library. This is because the test SDK automatically injects a `Program` class at build time. It's not something you have to worry about in general, but you may have issues if you try to add your own Program.cs file to your test project.[1]

Before we go any further and create some useful tests, we'll run the test project as it is, using both Visual Studio and the .NET SDK tooling, to see the expected output.

[1] This isn't a common thing to do, but I've seen it used occasionally. I describe this issue in detail, and how to fix it, in my blog post, "Fixing the error 'Program has more than one entry point defined' for console apps containing xUnit tests," at http://mng.bz/w9q5.

23.2.2 Running tests with dotnet test

When you create a test app that uses the .NET Test SDK, you can run your tests either with Visual Studio or using the .NET CLI. In Visual Studio, you run tests by choosing Tests > Run > All Tests from the main menu, or by clicking Run All in the Test Explorer window, as shown in figure 23.2.

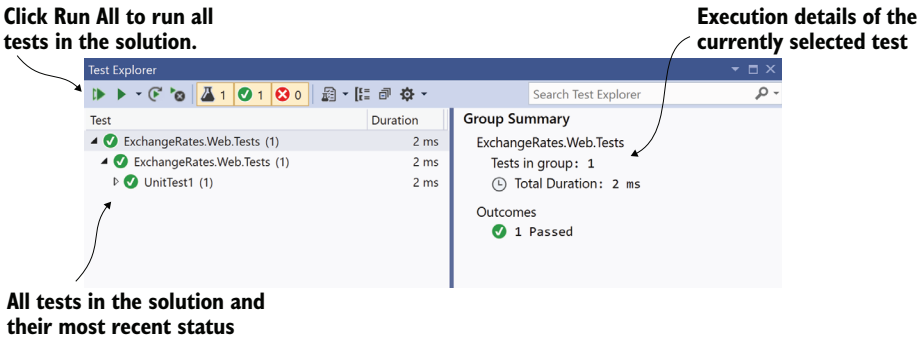


Figure 23.2 The Test Explorer window in Visual Studio lists all tests found in the solution and their most recent pass/fail status. Click a test in the left pane to see details about the most recent test run in the right pane.

The Test Explorer window lists all the tests found in your solution and the results of each test. In xUnit, a test will pass if it doesn't throw an exception, so `Test1` passed successfully.

Alternatively, you can run your tests from the command line using the .NET CLI by running

```
dotnet test
```

from the unit-test project's folder, as shown in figure 23.3.

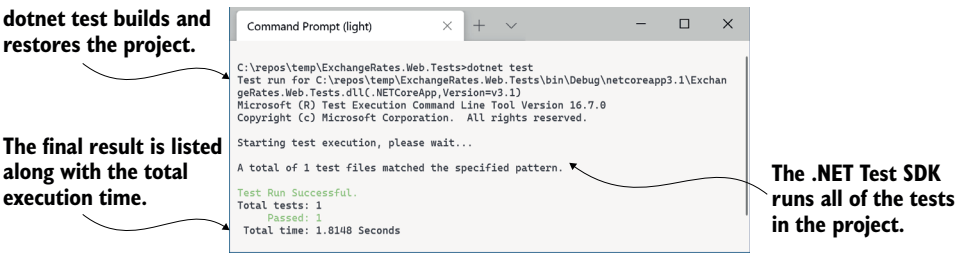


Figure 23.3 You can run tests from the command line using `dotnet test`. This restores and builds the test project before executing all the tests in the project.

NOTE You can also run `dotnet test` from the *solution* folder. This will run all test projects referenced in the .sln solution file.

Calling `dotnet test` runs a restore and build of your test project and then runs the tests, as you can see from the console output in figure 23.3. Under the hood, the .NET CLI calls into the same underlying infrastructure as Visual Studio does (the .NET SDK), so you can use whichever approach better suits your development style.

You've seen a successful test run, so it's time to replace that placeholder test with something useful. First things first, though; you need something to test.

23.2.3 Referencing your app from your test project

In test-driven development (TDD), you typically write your unit tests before you write the actual class you're testing, but I'm going to take a more traditional route here and create the class to test first. You'll write the tests for it afterwards.

Let's assume you've created an app called ExchangeRates.Web, which is used to convert between different currencies, and you want to add tests for it. You've added a test project to your solution as described in section 23.2.1, so your solution looks like figure 23.4.

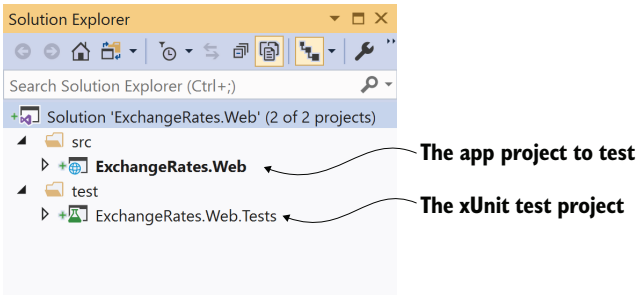


Figure 23.4 A basic solution containing an ASP.NET Core app called ExchangeRates.Web and a test project called ExchangeRates.Web.Tests.

In order for the ExchangeRates.Web.Tests project to be able to test the classes in the ExchangeRates.Web project, you need to add a reference to the web project in your test project. In Visual Studio, you can do this by right-clicking the Dependencies node of your test project and choosing Add Reference, as shown in figure 23.5. You can then select the web project from the Add Reference dialog box. After adding it to your project, it shows up inside the Dependencies node, under Projects.

Alternatively, you can edit the .csproj file directly and add a `<ProjectReference>` element inside an `<ItemGroup>` element with the relative path to the referenced project's .csproj file.

```
<ItemGroup>
  <ProjectReference
    Include="..\..\src\ExchangeRates.Web\ExchangeRates.Web.csproj" />
</ItemGroup>
```

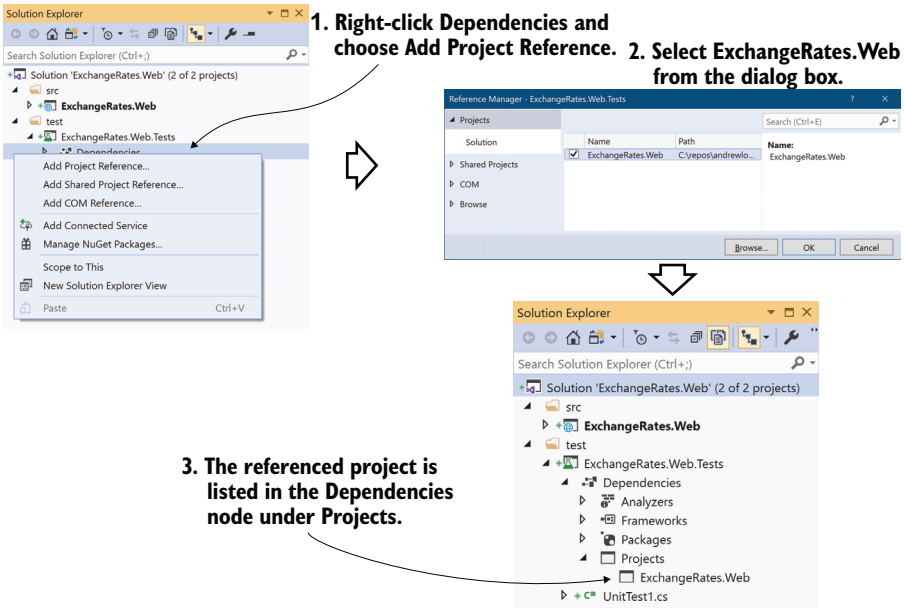


Figure 23.5 To test your app project, you need to add a reference to it from the test project. Right-click the Dependencies node and choose Add Project Reference. The app project is shown referenced inside the Dependencies node, under Projects.

Note that the path is the *relative* path. A "..." in the path means the parent folder, so the relative path shown correctly traverses the directory structure for the solution, including both the src and test folders shown in Solution Explorer in figure 23.5.

TIP Remember, you can edit the .csproj file directly in Visual Studio by double-clicking the project in Solution Explorer.

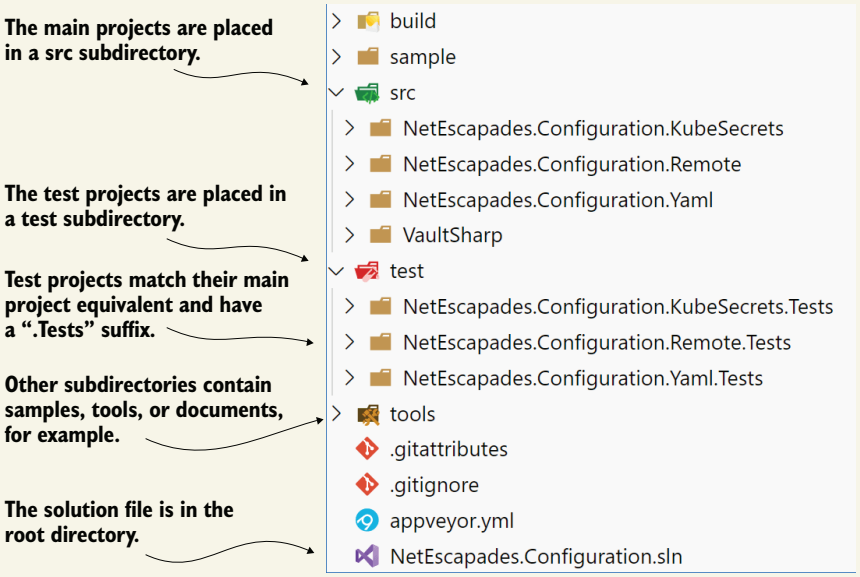
Common conventions for project layout

The layout and naming of projects within a solution is completely up to you, but ASP.NET Core projects have generally settled on a couple of conventions that differ slightly from the Visual Studio File > New defaults. These conventions are used by the ASP.NET team on GitHub, as well as by many other open source C# projects.

The following figure shows an example of these layout conventions. In summary, these are as follows:

- The .sln solution file is in the root directory.
- The main projects are placed in a src subdirectory.
- The test projects are placed in a test or tests subdirectory.
- Each main project has a test project equivalent, named the same as the associated main project with a ".Test" or ".Tests" suffix.

- Other folders, such as samples, tools, or docs contain sample projects, tools for building the project, or documentation.



Conventions around project structures have emerged in the ASP.NET Core framework libraries and open source projects on GitHub. You don't have to follow them for your own project, but it's worth being aware of them.

Whether or not you choose to follow these conventions is entirely up to you, but it's good to be aware of them at least, so you can easily navigate other projects on GitHub.

Your test project is now referencing your web project, so you can write tests for classes in the web project. You're going to be testing a simple class used for converting between currencies, as shown in the following listing.

Listing 23.3 Example `CurrencyConverter` class to convert currencies to GBP

```
public class CurrencyConverter
{
    public decimal ConvertToGbp(
        decimal value, decimal exchangeRate, int decimalPlaces)
    {
        if (exchangeRate <= 0)
        {
            throw new ArgumentException(
               "Exchange rate must be greater than zero",
               nameof(exchangeRate));
        }
```

The ConvertToGbp method converts a value using the provided exchange rate and rounds it.

Guard clause, as only positive exchange rates are valid

```
        var valueInGbp = value / exchangeRate;
        return decimal.Round(valueInGbp, decimalPlaces);
    }
}
```

Converts the value

Rounds the result and returns it

This class only has a single method, `ConvertToGbp()`, which converts a `value` from one currency into GBP, given the provided `exchangeRate`. It then rounds the value to the required number of decimal places and returns it.

> **WARNING** This class is only a basic implementation. In practice, you'd need to handle arithmetic overflow/underflow for large or negative values, as well as considering other edge cases. This is only for demonstration purposes!

Imagine you want to convert 5.27 USD to GBP, and the exchange rate from GBP to USD is 1.31. If you want to round to four decimal places, you'd make this call:

```
converter.ConvertToGbp(value: 5.27, exchangeRate: 1.31, decimalPlaces: 4);
```

You have your sample application, a class to test, and a test project, so it's about time you wrote some tests.

23.2.4 Adding Fact and Theory unit tests

When I write unit tests, I usually target one of three different paths through the method under test:

- *The happy path*—Where typical arguments with expected values are provided
- *The error path*—Where the arguments passed are invalid and tested for
- *Edge cases*—Where the provided arguments are right on the edge of expected values

I realize this is a broad classification, but it helps me think about the various scenarios I need to consider.[2] Let's start with the happy path, by writing a unit test that verifies that the `ConvertToGbp()` method is working as expected with typical input values.

Listing 23.4 Unit test for `ConvertToGbp` using expected arguments

```
[Fact]
public void ConvertToGbp_ConvertsCorrectly()
{
    var converter = new CurrencyConverter();
    decimal value = 3;
    decimal rate = 1.5m;
    int dp = 4;
    decimal expected = 2;
```

The [Fact] attribute marks the method as a test method.

You can call the test anything you like.

The class to test, commonly called the "system under test" (SUT)

The parameters of the test that will be passed to ConvertToGbp

The result you expect

[2] A whole other way to approach testing is property-based testing. This fascinating approach is common in functional programming communities, like F#. You can find a great introduction by Scott Wlaschin in his blog post, "An introduction to property-based testing": http://mng.bz/e5j9. That post uses F#, but it is still highly accessible even if you're new to the language.

```
    var actual = converter.ConvertToGbp(value, rate, dp);

    Assert.Equal(expected, actual);
}
```

Executes the method and captures the result

Verifies that the expected and actual values match. If they don't, this will throw an exception.

This is your first proper unit test, which has been configured using the Arrange, Act, Assert (AAA) style:

- *Arrange*—Define all the parameters and create an instance of the system (class) under test (SUT).
- *Act*—Execute the method being tested, and capture the result.
- *Assert*—Verify that the result of the Act stage had the expected value.

Most of the code in this test is standard C#, but if you're new to testing, the `Assert` call will be unfamiliar. This is a helper class provided by xUnit for making assertions about your code. If the parameters provided to `Assert.Equal()` aren't equal, the `Equal()` call will throw an exception and fail the test. If you change the `expected` variable in listing 23.4 to be `2.5` instead of `2`, for example, and run the test, you can see that Test Explorer shows a failure, as in figure 23.6.

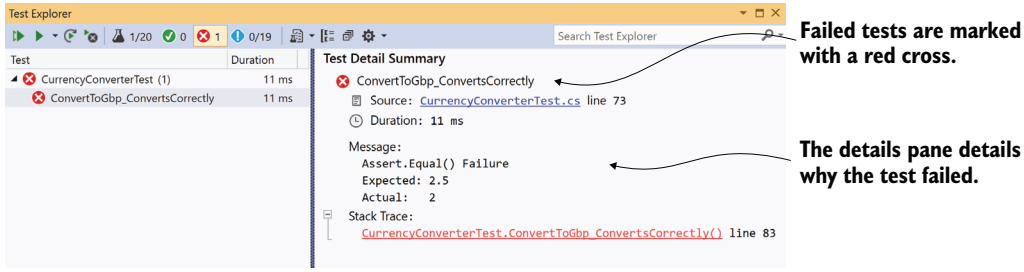


Figure 23.6 When a test fails, it's marked with a red cross in Test Explorer. Clicking the test in the left pane shows the reason for the failure in the right pane. In this case, the expected value was `2.5`, but the actual value was `2`.

> **TIP** Alternative assertion libraries such as Fluent Assertions (https://fluentassertions.com/) and Shouldly (https://github.com/shouldly/shouldly) allow you to write your assertions in a more natural style, such as `actual.Should().Be(expected)`. These libraries are entirely optional, but I find they make tests more readable and error messages easier to understand.

In listing 23.4 you chose specific values for `value`, `exchangeRate`, and `decimalPlaces` to test the happy path. But this is only one set of values in an infinite number of possibilities, so you should probably test at least *a few* different combinations.

One way to achieve this would be to copy and paste the test multiple times, tweak the parameters, and change the test method name to make it unique. xUnit provides an alternative way to achieve the same thing without requiring so much duplication.

> **NOTE** The names of your test class and method are used throughout the test framework to describe your test. You can customize how these are displayed in Visual Studio and in the CLI by configuring an xunit.runner.json file, as described here: https://xunit.net/docs/configuration-files.

Instead of creating a `[Fact]` test method, you can create a `[Theory]` test method. A theory provides a way of parameterizing your test methods, effectively taking your test method and running it multiple times with different arguments. Each set of arguments is considered a different test.

You could rewrite the `[Fact]` test in listing 23.4 to be a `[Theory]` test, as shown next. Instead of specifying the variables in the method body, pass them as parameters to the method, and then decorate the method with three `[InlineData]` attributes. Each instance of the attribute provides the parameters for a single run of the test.

Listing 23.5 Theory test for `ConvertToGbp` testing multiple sets of values

```
[Theory]
[InlineData(0, 3, 0)]
[InlineData(3, 1.5, 2)]
[InlineData(3.75, 2.5, 1.5)]
public void ConvertToGbp_ConvertsCorrectly (
    decimal value, decimal rate, decimal expected)
{
    var converter = new CurrencyConverter();
    int dps = 4;

    var actual = converter.ConvertToGbp(value, rate, dps);

    Assert.Equal(expected, actual);
}
```

Marks the method as a parameterized test

Each [InlineData] attribute provides all the parameters for a single run of the test method.

The method takes parameters, which are provided by the [InlineData] attributes.

Executes the system under test

Verifies the result

If you run this `[Theory]` test using `dotnet test` or Visual Studio, it will show up as three separate tests, one for each set of `[InlineData]`, as shown in figure 23.7.

`[InlineData]` isn't the only way to provide the parameters for your theory tests, but it's one of the most commonly used. You can also use a static property on your test class with the `[MemberData]` attribute, or a class itself using the `[ClassData]` attribute.[3]

You now have some tests for the happy path of the `ConvertToGbp()` method, and I even sneaked an edge case into listing 23.5 by testing the case where `value = 0`. The final concept I'll cover is testing error cases, where invalid values are passed to the method under test.

---

[3] I describe how you can use the `[ClassData]` and `[MemberData]` attributes in a blog post, "Creating parameterised tests in xUnit with [InlineData], [ClassData], and [MemberData]": http://mng.bz/8ayP.

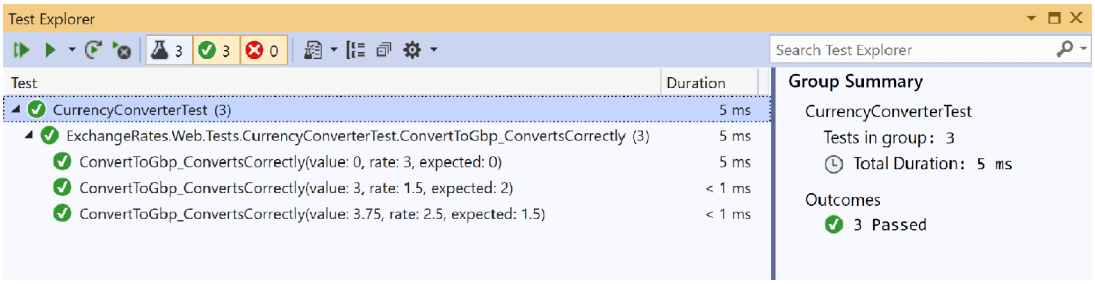


Figure 23.7 Each set of parameters in an `[InlineData]` attribute for a `[Theory]` test creates a separate test run. In this example, a single `[Theory]` has three `[InlineData]` attributes, so it creates three tests, named according to the method name and the provided parameters.

23.2.5 Testing failure conditions

A key part of unit testing is checking that the system under test handles edge cases and errors correctly. For the `CurrencyConverter`, that would mean checking how the class handles negative values, small or zero exchange rates, large values and rates, and so on.

Some of these edge cases might be rare but valid cases, whereas other cases might be technically invalid. Calling `ConvertToGbp` with a negative `value` is probably valid; the converted result should be negative too. A negative exchange rate doesn't make sense conceptually, so it should be considered an invalid value.

Depending on the design of the method, it's common to throw exceptions when invalid values are passed to a method. In listing 23.3 you saw that we throw an `ArgumentException` if the `exchangeRate` parameter is less than or equal to `0`.

xUnit includes a variety of helpers on the `Assert` class for testing whether a method throws an exception of an expected type. You can then make further assertions on the exception; for example, to test whether the exception had an expected message.

> **WARNING** Take care not to tie your test methods too closely to the internal implementation of a method. Doing so can make your tests brittle, where trivial changes to a class break the unit tests.

The following listing shows a `[Fact]` test to check the behavior of the `ConvertToGbp()` method when you pass it a `0` `exchangeRate`. The `Assert.Throws` method takes a lambda function that describes the action to execute, which should throw an exception when run.

Listing 23.6 Using `Assert.Throws<>` to test whether a method throws an exception

```
[Fact]
public void ThrowsExceptionIfRateIsZero()
{
    var converter = new CurrencyConverter();
    const decimal value = 1;
    const decimal rate = 0;        <-- An invalid value
```

```
    const int dp = 2;
    var ex = Assert.Throws<ArgumentException>(
        () => converter.ConvertToGbp(value, rate, dp));

    // Further assertions on the exception thrown, ex
}
```

You expect an ArgumentException to be thrown.

The method to execute, which should throw an exception

The `Assert.Throws` method executes the lambda and catches the exception. If the exception thrown matches the expected type, the test will pass. If no exception is thrown or the exception thrown isn't of the expected type, the `Assert.Throws` method will throw an exception and fail the test.

That brings us to the end of this introduction on unit testing with xUnit. The examples in this section described how to use the new .NET Test SDK, but we didn't cover anything specific to ASP.NET Core. In the rest of this chapter we'll focus on testing ASP.NET Core projects specifically. We'll start by unit testing middleware.

23.3 Unit testing custom middleware

In this section you'll learn how to test custom middleware in isolation. You'll see how to test whether your middleware handled a request or whether it called the next middleware in the pipeline. You'll also see how to read the response stream for your middleware.

In chapter 19 you saw how to create custom middleware and how you could encapsulate middleware as a class with an `Invoke` function. In this section you'll create unit tests for a simple health-check middleware component, similar to the one in chapter 19. This is a basic implementation, but it demonstrates the approach you can take for more complex middleware components.

The middleware you'll be testing is shown in listing 23.7. When invoked, this middleware checks that the path starts with `/ping` and, if it does, returns a plain text `"pong"` response. If the request doesn't match, it calls the next middleware in the pipeline (the provided `RequestDelegate`).

Listing 23.7 `StatusMiddleware` to be tested, which returns a `"pong"` response

```
public class StatusMiddleware
{
    private readonly RequestDelegate _next;
    public StatusMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task Invoke(HttpContext context)
    {
        if(context.Request.Path.StartsWithSegments("/ping"))
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
            return;
        }
```

The RequestDelegate representing the rest of the middleware pipeline

Called when the middleware is executed

If the path starts with "/ping", a "pong" response is returned.

```
        await _next(context);
    }
}
```

Otherwise, the next middleware in the pipeline is invoked.

In this section, you're only going to test two simple cases:

- When a request is made with a path of `"/ping"`
- When a request is made with a different path

WARNING Where possible, I recommend you *don't* directly inspect paths in your middleware like this. A better approach is to use endpoint routing instead, as I discussed in chapter 19. The middleware in this section is for demonstration purposes only.

Middleware is slightly complicated to unit test because the `HttpContext` object is conceptually a *big* class. It contains all the details for the request and the response, which can mean there's a lot of surface area for your middleware to interact with. For that reason, I find unit tests tend to be tightly coupled to the middleware implementation, which is generally undesirable.

For the first test, you'll look at the case where the incoming request `Path` *doesn't* start with `/ping`. In this case, `StatusMiddleware` should leave the `HttpContext` unchanged and should call the `RequestDelegate` provided in the constructor, which represents the next middleware in the pipeline.

You could test this behavior in several ways, but in listing 23.8 you test that the `RequestDelegate` (essentially a one-parameter function) is executed by setting a local variable to `true`. In the `Assert` at the end of the method, you verify that the variable was set and therefore that the delegate was invoked. To invoke `StatusMiddleware`, create and pass in a `DefaultHttpContext`,[4] which is an implementation of `HttpContext`.

Listing 23.8 Unit testing `StatusMiddleware` when a nonmatching `path` is provided

```
[Fact]
public async Task ForNonMatchingRequest_CallsNextDelegate()
{
    var context = new DefaultHttpContext();
    context.Request.Path = "/somethingelse";
    var wasExecuted = false;
    RequestDelegate next = (HttpContext ctx) =>
    {
        wasExecuted = true;
        return Task.CompletedTask;
    };
    var middleware = new StatusMiddleware(next);
```

Creates a DefaultHttpContext and sets the path for the request

Tracks whether the RequestDelegate was executed

The RequestDelegate representing the next middleware should be invoked in this example.

Creates an instance of the middleware, passing in the next RequestDelegate

[4] The `DefaultHttpContext` derives from `HttpContext` and is part of the base ASP.NET Core framework abstractions. If you're so inclined, you can explore the source code for it on GitHub at http://mng.bz/q9qx.

```
    await middleware.Invoke(context);

    Assert.True(wasExecuted);
}
```

Invokes the middleware with the HttpContext; should invoke the RequestDelegate

Verifies RequestDelegate was invoked

When the middleware is invoked, it checks the provided `Path` and finds that it doesn't match the required value of `/ping`. The middleware therefore calls the next `RequestDelegate` and returns.

The other obvious case to test is when the request `Path` is `"/ping"`; the middleware should generate an appropriate response. You could test several different characteristics of the response:

- The response should have a `200 OK` status code.
- The response should have a `Content-Type` of `text/plain`.
- The response body should contain the `"pong"` string.

Each of these characteristics represents a different requirement, so you'd typically codify each as a separate unit test. This makes it easier to tell exactly which requirement hasn't been met when a test fails. For simplicity, in listing 23.9 I show all these assertions in the same test.

The positive case unit test is made more complex by the need to read the response body to confirm it contains `"pong"`. `DefaultHttpContext` uses `Stream.Null` for the `Response.Body` object, which means anything written to `Body` is lost. To capture the response and read it out to verify the contents, you must replace the `Body` with a `MemoryStream`. After the middleware executes, you can use a `StreamReader` to read the contents of the `MemoryStream` into a `string` and verify it.

Listing 23.9 Unit testing `StatusMiddleware` when a matching `Path` is provided

```
[Fact]
public async Task ReturnsPongBodyContent()
{
    var bodyStream = new MemoryStream();
    var context = new DefaultHttpContext();
    context.Response.Body = bodyStream;
    context.Request.Path = "/ping";
    RequestDelegate next = (ctx) => Task.CompletedTask;
    var middleware = new StatusMiddleware(next: next);

    await middleware.Invoke(context);

    string response;
    bodyStream.Seek(0, SeekOrigin.Begin);
    using (var stringReader = new StreamReader(bodyStream))
    {
        response = await stringReader.ReadToEndAsync();
    }
```

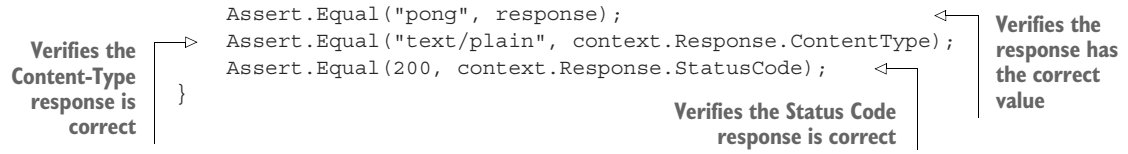
Creates a DefaultHttpContext and initializes the body with a MemoryStream to capture the response

The path is set to the required value for the StatusMiddleware.

Creates an instance of the middleware and passes in a simple RequestDelegate

Invokes the middleware

Rewinds the MemoryStream and reads the response body into a string



As you can see, unit testing middleware requires a lot of setup to get it working. On the positive side, it allows you to test your middleware in isolation, but in some cases, especially for simple middleware without any dependencies on databases or other services, integration testing can (somewhat surprisingly) be easier. In section 23.5 you'll create integration tests for this middleware to see the difference.

Custom middleware is common in ASP.NET Core projects, but far more common are Razor Pages and API controllers. In the next section you'll see how you can unit test them in isolation from other components.

23.4 Unit testing API controllers

In this section you'll learn how to unit test API controllers. You'll learn about the benefits and difficulties of testing these components in isolation, and the situations when it can be useful.

Unit tests are all about isolating behavior; you want to test only the logic contained in the component itself, separate from the behavior of any dependencies. The Razor Pages and MVC/API *frameworks* use the filter pipeline, routing, and model-binding systems, but these are all *external* to the controller or `PageModels`. The `PageModels` and controllers themselves are responsible for only a limited number of things. Typically,

- For invalid requests (that have failed validation, for example), return an appropriate `ActionResult` (API controllers) or redisplay a form (Razor Pages).
- For valid requests, call the required business logic services and return an appropriate `ActionResult` (API controllers), or show or redirect to a success page (Razor Pages).
- Optionally, apply resource-based authorization as required.

Controllers and Razor Pages generally shouldn't contain business logic themselves; instead, they should call out to other services. Think of them more as orchestrators, serving as the intermediary between the HTTP interfaces your app exposes and your business logic services.

If you follow this separation, you'll find it easier to write unit tests for your business logic, and you'll benefit from greater flexibility when you want to change your controllers to meet your needs. With that in mind, there's often a drive to make your controllers and page handlers as thin as possible,[5] to the point where there's not much left to test!

[5] One of my first introductions to this idea was a series of posts by Jimmy Bogard. The following link points to the last post in the series, but it contains links to all the earlier posts too. Jimmy Bogard is also behind the MediatR library (https://github.com/jbogard/MediatR), which makes creating thin controllers even easier. See "Put your controllers on a diet: POSTs and commands": http://mng.bz/7VNQ.

All that said, controllers and actions are classes and methods, so you *can* write unit tests for them. The difficulty is deciding what you want to test. As an example, we'll consider the simple API controller in the following listing, which converts a value using a provided exchange rate and returns a response.

Listing 23.10 The API controller under test

```
[Route("api/[controller]")]
public class CurrencyController : ControllerBase
{
    private readonly CurrencyConverter _converter
       = new CurrencyConverter();

    [HttpPost]
    public ActionResult<decimal> Convert(InputModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        decimal result = _convert.ConvertToGbp(model)

        return result;
    }
}
```

The CurrencyConverter would normally be injected using DI. Created here for simplicity

The Convert method returns an ActionResult<T>.

If the input is invalid, returns a 400 Bad Request result, including the ModelState.

If the model is valid, calculate the result.

Return the result directly.

Let's first consider the happy path, when the controller receives a valid request. The following listing shows that you can create an instance of the API controller, call an action method, and you'll receive an `ActionResult<T>` response.

Listing 23.11 A simple API controller unit test

```
public class CurrencyControllerTest
{
    [Fact]
    public void Convert_ReturnsValue()
    {
        var controller = new CurrencyController();
        var model = new ConvertInputModel
        {
            Value = 1,
            ExchangeRate = 3,
            DecimalPlaces = 2,
        };

        ActionResult<decimal> result = controller.Convert(model);
        Assert.NotNull(result);
    }
}
```

Creates an instance of the ConvertController to test and a model to send to the API

Asserts that the IActionResult is a ViewResult

Invokes the ConvertToGbp method and captures the value returned

An important point to note here is that you're *only* testing the return value of the action, the `ActionResult<T>`, *not* the response that's sent back to the user. The process of serializing the result to the response is handled by the MVC formatter infrastructure, as you saw in chapter 9, not by the controller.

When you unit test controllers, you're testing them *separately* from the MVC infrastructure, such as formatting, model-binding, routing, and authentication. This is obviously by design, but as with testing middleware in section 23.3, it can make testing some aspects of your controller somewhat complex.

Consider model validation. As you saw in chapter 6, one of the key responsibilities of action methods and Razor Page handlers is to check the `ModelState.IsValid` property and act accordingly if a binding model is invalid. Testing that your controllers and `PageModels` correctly handle validation failures seems like a good candidate for a unit test.

Unfortunately, things aren't simple here either. The Razor Page/MVC framework automatically sets the `ModelState` property as part of the model-binding process. In practice, when your action method or page handler is invoked in your running app, you know that the `ModelState` will match the binding model values. But in a unit test, there's no model-binding, so you must set the `ModelState` yourself manually.

Imagine you're interested in testing the error path for the controller in listing 23.10, where the model is invalid and the controller should return `BadRequestObjectResult`. In a unit test, you can't rely on the `ModelState` property being correct for the binding model. Instead, you must *manually* add a model-binding error to the controller's `ModelState` before calling the action, as shown here.

Listing 23.12 Testing handling of validation errors in MVC controllers

```
[Fact]
public void Convert_ReturnsBadRequestWhenInvalid()
{
    var controller = new CurrencyController();
    var model = new ConvertInputModel
    {
        Value = 1,
        ExchangeRate = -2,
        DecimalPlaces = 2,
    };

    controller.ModelState.AddModelError(
        nameof(model.ExchangeRate),
        "Exchange rate must be greater than zero"
    );

    ActionResult<decimal> result = controller.Convert(model);

    Assert.IsType<BadRequestObjectResult>(result.Result);
}
```

Creates an instance of the Controller to test

Creates an invalid binding model by using a negative ExchangeRate

Manually adds a model error to the Controller's ModelState. This sets ModelState.IsValid to false.

Invokes the action method, passing in the binding models

Verifies the action method returned a BadRequestObjectResult

> **NOTE** In listing 23.12, I passed in an invalid model, but I could just as easily have passed in a *valid* model, or even `null`; the controller doesn't use the binding model if the `ModelState` isn't valid, so the test would still pass. But if you're writing unit tests like this one, I recommend trying to keep your model consistent with your `ModelState`; otherwise your unit tests aren't testing a situation that occurs in practice.

Personally, I tend to shy away from unit testing API controllers directly in this way.[6] As you've seen with model binding, the controllers are somewhat dependent on earlier stages of the MVC framework, which you often need to emulate. Similarly, if your controllers access the `HttpContext` (available on the `ControllerBase` base classes), you may need to perform additional setup.

> **NOTE** I haven't discussed Razor Pages much in this section, as they suffer from many of the same problems, in that they are dependent on the supporting infrastructure of the framework. Nevertheless, if you do wish to test your Razor Page `PageModel`, you can read about it in Microsoft's "Razor Pages unit tests in ASP.NET Core" documentation: http://mng.bz/GxmM.

Instead of using unit testing, I try to keep my controllers and Razor Pages as "thin" as possible. I push as much of the behavior in these classes into business logic services that can be easily unit tested, or into middleware and filters, which can be more easily tested independently.

> **NOTE** This is a personal preference. Some people like to get as close to 100% test coverage for their code base as possible, but I find testing "orchestration" classes is often more hassle than it's worth.

Although I often forgo *unit* testing controllers and Razor Pages, I often write *integration* tests that test them in the context of a complete application. In the next section, we'll look at ways to write integration tests for your app, so you can test its various components in the context of the ASP.NET Core framework as a whole.

23.5 Integration testing: Testing your whole app in-memory

In this section you'll learn how to create integration tests that test component interactions. You'll learn to create a `TestServer` that sends HTTP requests in-memory to test custom middleware components more easily. You'll then learn how to run integration tests for a real application, using your real app's configuration, services, and middleware pipeline. Finally, you'll learn how to use `WebApplicationFactory` to replace services in your app with test versions, to avoid depending on third-party APIs in your tests.

If you search the internet for the different types of testing, you'll find a host of different types to choose from. The differences between them are sometimes subtle, and

[6] You can read more about why I generally don't unit test my controllers in my blog article, "Should you unit-test API/MVC controllers in ASP.NET Core?": http://mng.bz/YqMo.

people don't universally agree upon the definitions. I chose not to dwell on it in this book—I consider unit tests to be isolated tests of a component and integration tests to be tests that exercise multiple components at once.

In this section I'm going to show how you can write integration tests for the `StatusMiddleware` from section 23.3 and the API controller from section 23.4. Instead of isolating the components from the surrounding framework and invoking them directly, you'll specifically test them in a context similar to how you use them in practice.

Integration tests are an important part of confirming that your components function correctly, but they don't remove the need for unit tests. Unit tests are excellent for testing small pieces of logic contained in your components and are typically quick to execute. Integration tests are normally significantly slower, as they require much more configuration and may rely on external infrastructure, such as a database.

Consequently, it's normal to have far more unit tests for an app than integration tests. As you saw in section 23.2, unit tests typically verify the behavior of a component, using valid inputs, edge cases, and invalid inputs to ensure that the component behaves correctly in all cases. Once you have an extensive suite of unit tests, you'll likely only need a few integration tests to be confident your application is working correctly.

You could write many different types of integration tests for an application. You could test that a service can write to a database correctly, that it can integrate with a third-party service (for sending emails, for example), or that it can handle HTTP requests made to it.

In this section we're going to focus on the last point, verifying that your app can handle requests made to it, just as it would if you were accessing the app from a browser. For this, we're going to use a useful library provided by the ASP.NET Core team called Microsoft.AspNetCore.TestHost.

23.5.1 Creating a TestServer using the Test Host package

Imagine you want to write some integration tests for the `StatusMiddleware` from section 23.3. You've already written unit tests for it, but you want to have at least one integration test that tests the middleware in the context of the ASP.NET Core infrastructure.

You could go about this in many ways. Perhaps the most complete approach would be to create a separate project and configure `StatusMiddleware` as the only middleware in the pipeline. You'd then need to run this project, wait for it to start up, send requests to it, and inspect the responses.

This would possibly make for a good test, but it would also require a lot of configuration, and it would be fragile and error prone. What if the test app can't start because it tries to use an already-taken port? What if the test app doesn't shut down correctly? How long should the integration test wait for the app to start?

The ASP.NET Core Test Host package lets you get close to this setup without having the added complexity of spinning up a separate app. You add the Test Host to your test project by adding the Microsoft.AspNetCore.TestHost NuGet package, either

using the Visual Studio NuGet GUI, Package Manager Console, or .NET CLI. Alternatively, add the `<PackageReference>` element directly to your test project's .csproj file:

```
<PackageReference Include="Microsoft.AspNetCore.TestHost" Version="5.0.0"/>
```

In a typical ASP.NET Core app, you create a `HostBuilder` in your `Program` class, configure a web server (Kestrel), and define your application's configuration, services, and middleware pipeline (using a Startup file). Finally, you call `Build()` on the `HostBuilder` to create an instance of an `IHost` that can be run and that will listen for requests on a given URL and port.

The Test Host package uses the same `HostBuilder` to define your test application, but instead of listening for requests at the network level, it creates an `IHost` that uses in-memory request objects instead, as shown in figure 23.8. It even exposes an

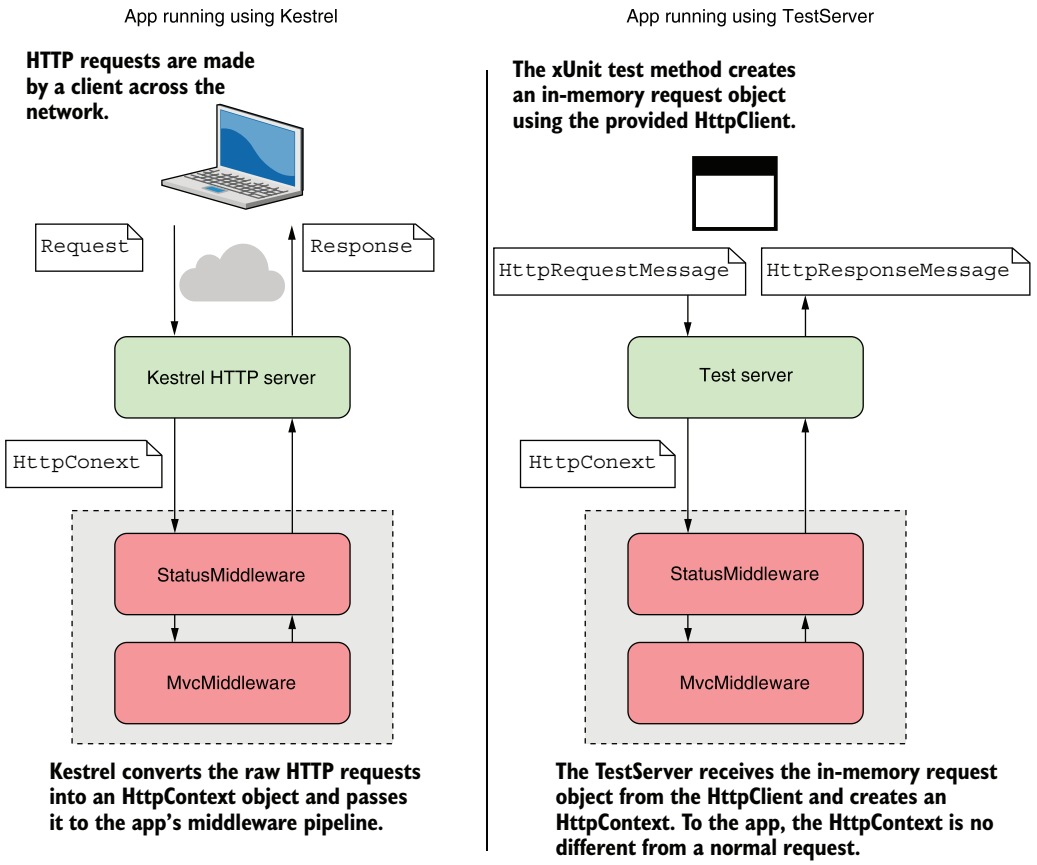


Figure 23.8 When your app runs normally, it uses the Kestrel server. This listens for HTTP requests and converts the requests into an `HttpContext`, which is passed to the middleware pipeline. The `TestServer` doesn't listen for requests on the network. Instead, you use an `HttpClient` to make in-memory requests. From the point of view of the middleware, there's no difference.

`HttpClient` that you can use to send requests to the test app. You can interact with the `HttpClient` as though it were sending requests over the network, but in reality the requests are kept entirely in memory.

Listing 23.13 shows how to use the Test Host package to create a simple integration test for the `StatusMiddleware`. First, create a `HostBuilder` and call `ConfigureWebHost()` to define your application by adding middleware in the `Configure` method. This is equivalent to the `Startup.Configure()` method you would typically use to configure your application.

Call the `UseTestServer()` extension method in `ConfigureWebHost()`, which replaces the default Kestrel server with the `TestServer` from the Test Host package. The `TestServer` is the main component in the Test Host package, which makes all the magic possible. After configuring the `HostBuilder`, call `StartAsync()` to build and start the test application. You can then create an `HttpClient` using the extension method `GetTestClient()`. This returns an `HttpClient` configured to make in-memory requests to the `TestServer`.

Listing 23.13 Creating an integration test with `TestServer`

```
public class StatusMiddlewareTests
{
    [Fact]
    public async Task StatusMiddlewareReturnsPong()
    {
        var hostBuilder = new HostBuilder()
            .ConfigureWebHost(webHost =>
            {
                webHost.Configure(app =>
                    app.UseMiddleware<StatusMiddleware>());
                webHost.UseTestServer();
            });

        IHost host = await hostBuilder.StartAsync();
        HttpClient client = host.GetTestClient();

        var response = await client.GetAsync("/ping");

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();
        Assert.Equal("pong", content);
    }
}
```

Configures a HostBuilder to define the in-memory test app

Add the StatusMiddleware as the only middleware in the pipeline.

Configure the host to use the TestServer instead of Kestrel.

Build and start the host.

Creates an HttpClient, or you can interact directly with the server object

Makes an in-memory request, which is handled by the app as normal

Verifies the response was a success (2xx) status code

Reads the body content and verifies that it contained "pong"

This test ensures that the test application defined by `HostBuilder` returns the expected value when it receives a request to the `/ping` path. The request is entirely in-memory, but from the point of view of `StatusMiddleware`, it's the same as if the request came from the network.

The `HostBuilder` configuration in this example is simple. Even though I've called this an integration test, you're specifically testing the `StatusMiddleware` on its own,

rather than in the context of a "real" application. In many ways, I think this setup is preferable for testing custom middleware compared to the "proper" unit tests I showed in section 23.3.

Regardless of what you call it, this test relies on very simple configuration for the test app. You may also want to test the middleware in the context of your *real* application, so that the result is representative of your app's *real* configuration.

If you want to run integration tests based on an existing app, then you won't want to have to configure the test `HostBuilder` manually like you did in listing 23.13. Instead, you can use another helper package, Microsoft.AspNetCore.Mvc.Testing.

23.5.2 Testing your application with WebApplicationFactory

Building up a `HostBuilder` and using the Test Host package, as you did in section 23.5.1, can be useful when you want to test isolated "infrastructure" components, such as middleware. It's also very common to want to test your "real" app, with the full middleware pipeline configured and all the required services added to DI. This gives you the most confidence that your application is going to work in production.

The `TestServer` that provides the in-memory server can be used for testing your real app, but, in principle, there's a lot more configuration required. Your real app likely loads configuration files or static files, and it may use Razor Pages and views. Prior to .NET Core 2.1, configuring all of these was cumbersome. Thankfully, the introduction of the Microsoft.AspNetCore.Mvc.Testing package and `WebApplicationFactory` largely solves these configuration issues for you.

You can use the `WebApplicationFactory` class (provided by the Microsoft.AspNetCore.Mvc.Testing NuGet package) to run an in-memory version of your real application. It uses the `TestServer` behind the scenes, but it uses your app's real configuration, DI service registration, and middleware pipeline. For example, the following listing shows an example that tests that when your application receives a `"/ping"` request, it responds with `"pong"`.

Listing 23.14 Creating an integration test with `WebApplicationFactory`

```
public class IntegrationTests:
    IClassFixture<WebApplicationFactory<Startup>>
{
    private readonly WebApplicationFactory<Startup> _fixture;
    public IntegrationTests(
        WebApplicationFactory<Startup> fixture)
    {
        _fixture = fixture;
    }

    [Fact]
    public async Task PingRequest_ReturnsPong()
    {
        HttpClient client = _fixture.CreateClient();
```

Your test must implement the interface, though there are no methods to implement.

Inject an instance of WebApplicationFactory<T>, where T is a class in your app.

Create an HttpClient that sends requests to the in-memory TestServer.

```
        var response = await client.GetAsync("/ping");

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();
        Assert.Equal("pong", content);
    }
}
```

Make requests and verify the response as before.

One of the advantages of using `WebApplicationFactory` as shown in listing 23.14 is that it requires *less* manual configuration than using the `TestServer` directly, as shown in listing 23.13, despite performing *more* configuration behind the scenes. The `WebApplicationFactory` tests your app using the configuration defined in your Program.cs and Startup.cs files.

Listings 23.14 and 23.13 are *conceptually* quite different too. Listing 23.13 tests that the `StatusMiddleware` behaves as expected in the context of a dummy ASP.NET Core app; listing 23.14 tests that *your app behaves as expected for a given input*. It doesn't say anything specific about *how* that happens. Your app doesn't have to use the `StatusMiddleware` for the test in listing 23.14 to pass; it just has to respond correctly to the given request. That means the test knows less about the internal implementation details of your app and is only concerned with its *behavior*.

DEFINITION Tests that fail whenever you change your app slightly are called *brittle* or *fragile*. Try to avoid brittle tests by ensuring they aren't dependent on the implementation details of your app.

To create tests that use `WebApplicationFactory`:

1. Install the Microsoft.AspNetCore.Mvc.Testing NuGet package in your project by running `dotnet add package Microsoft.AspNetCore.Mvc.Testing`, by using the NuGet explorer in Visual Studio, or by adding a `<PackageReference>` element to your project file as follows:

   ```
   <PackageReference Include="Microsoft.AspNetCore.Mvc.Testing"
       Version="5.0.0" />
   ```

2. Update the `<Project>` element in your test project's .csproj file to the following:

   ```
   <Project Sdk="Microsoft.NET.Sdk.Web">
   ```

   This is required by `WebApplicationFactory` so that it can find your configuration files and static files.

3. Implement `IClassFixture<WebApplicationFactory<T>>` in your xUnit test class, where `T` is a class in your real application's project. By convention, you typically use your application's `Startup` class for `T`.
   - `WebApplicationFactory` uses the `T` reference to find your application's `Program.CreateHostBuilder()` method to build an appropriate `TestServer` for tests.

  - The `IClassFixture<TFixture>` is an xUnit marker interface that tells xUnit to build an instance of `TFixture` before building the test class and to inject the instance into the test class's constructor. You can read more about fixtures at https://xunit.net/docs/shared-context.

4. Accept an instance of `WebApplicationFactory<T>` in your test class's constructor. You can use this fixture to create an `HttpClient` for sending in-memory requests to the `TestServer`. Those requests emulate your application's production behavior, as your application's real configuration, services, and middleware are all used.

The big advantage of `WebApplicationFactory` is that you can easily test your real app's behavior. That power comes with responsibility—your app will behave just as it would in real life, so it will write to a database and send to third-party APIs! Depending on what you're testing, you may want to replace some of your dependencies to avoid this, as well as to make testing easier.

23.5.3 Replacing dependencies in WebApplicationFactory

When you use `WebApplicationFactory` to run integration tests on your app, your app will be running in-memory, but other than that, it's as though you're running your application using `dotnet run`. That means any connection strings, secrets, or API keys that can be loaded locally will also be used to run your application.

> **TIP** By default, `WebApplicationFactory` uses the `"Development"` hosting environment, the same as when you run locally.

On the plus side, that means you have a genuine test that your application can start correctly. For example, if you've forgotten to register a required DI dependency that is detected on application startup, any tests that use `WebApplicationFactory` will fail.

On the downside, that means all your tests will be using the same database connection and services as when you run your application locally. It's common to want to replace those with alternative "test" versions of your services.

As a simple example, let's imagine the `CurrencyConverter` that you've been testing in this app uses `IHttpClientFactory` to call a third-party API to retrieve the latest exchange rates. You don't want to hit that API repeatedly in your integration tests, so you want to replace the `CurrencyConverter` with your own `StubCurrencyConverter`.

The first step is to ensure the service `CurrencyConverter` implements an interface, `ICurrencyConverter` for example, and that your app uses this interface throughout, not the implementation. For our simple example, the interface would probably look like the following:

```
public interface ICurrencyConverter
{
    decimal ConvertToGbp(decimal value, decimal rate, int dps);
}
```

You would register the service in `Startup.ConfigureServices()` using

```
services.AddScoped<ICurrencyConverter, CurrencyConverter>();
```

Now that your application only indirectly depends on `CurrencyConverter`, you can provide an alternative implementation in your tests.

> **TIP** Using an interface decouples your application services from a specific implementation, allowing you to substitute alternative implementations. This is a key practice for making classes testable.

We'll create a simple alternative implementation of `ICurrencyConverter` for our tests that always returns the same value, `3`. It's obviously not very useful as an actual converter, but that's not the point: you have complete control! Create the following class in your test project:

```
public class StubCurrencyConverter : ICurrencyConverter
{
    public decimal ConvertToGbp(decimal value, decimal rate, int dps)
    {
        return 3;
    }
}
```

You now have all the pieces you need to replace the implementation in your tests. To achieve that, we'll use a feature of `WebApplicationFactory` that lets you customize the DI container before starting the test server.

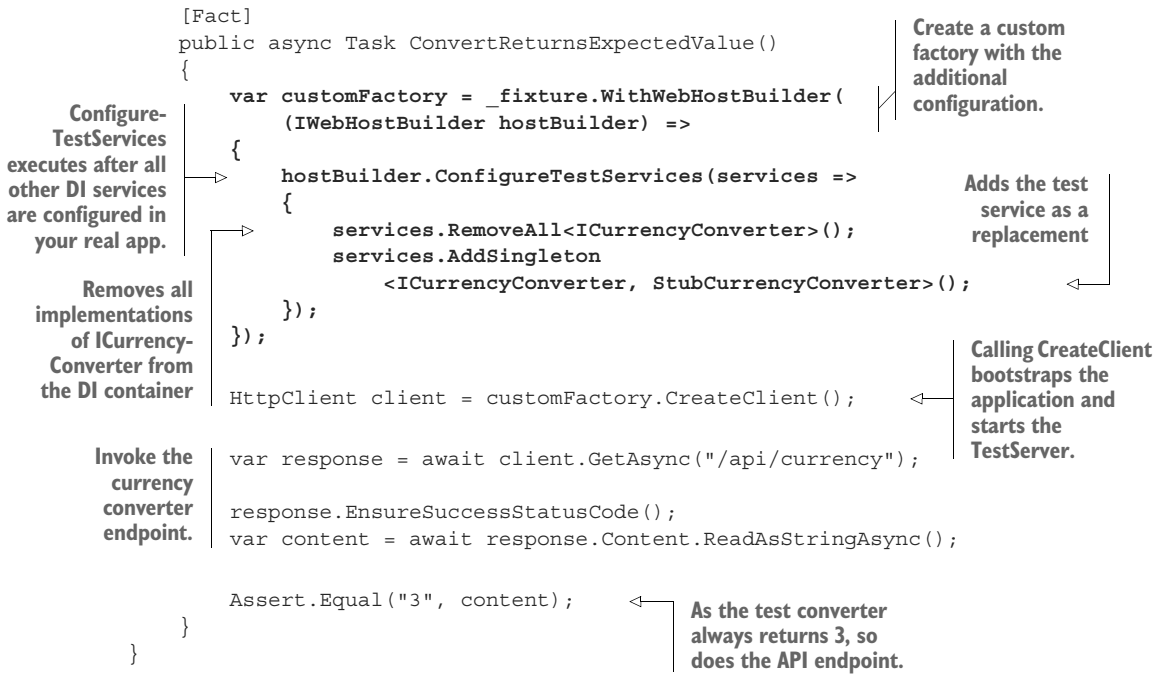
> **TIP** It's important to remember you *only* want to replace the implementation when running in the test project. I've seen some people try to configure their *real* apps to replace live services for fake services when a specific value is set, for example. That is generally unnecessary, bloats your apps with test services, and generally adds confusion!

`WebApplicationFactory` exposes a method, `WithWebHostBuilder`, that allows you to customize your application before the in-memory `TestServer` starts. The following listing shows an integration test that uses this builder to replace the "default" `ICurrencyConverter` implementation with our test stub.

Listing 23.15 Replacing a dependency in a test using `WithWebHostBuilder`

```
public class IntegrationTests:
    IClassFixture<WebApplicationFactory<Startup>>
{
    private readonly WebApplicationFactory<Startup> _fixture;
    public IntegrationTests(WebApplicationFactory<Startup> fixture)
    {
        _fixture = fixture;
    }
```

Implement the required interface, and inject it into the constructor.



There are a couple of important points to note in this example:

- `WithWebHostBuilder()` returns a *new* `WebApplicationFactory` instance. The new instance has your custom configuration, while the original injected `_fixture` instance remains unchanged.
- `ConfigureTestServices()` is called after your real app's `ConfigureServices()` method. That means you can replace services that have been previously registered. You can also use this to override configuration values, as you'll see in section 23.6.

`WithWebHostBuilder()` is handy when you want to replace a service for a single test. But what if you wanted to replace the `ICurrencyConverter` in every test. All that boilerplate would quickly become cumbersome. Instead, you can create a custom `WebApplicationFactory`.

23.5.4 Reducing duplication by creating a custom WebApplicationFactory

If you find yourself writing `WithWebHostBuilder()` a lot in your integration tests, it might be worth creating a custom `WebApplicationFactory` instead. The following listing shows how to centralize the test service we used in listing 23.15 into a custom `WebApplicationFactory`.

Listing 23.16 Creating a custom `WebApplicationFactory` to reduce duplication

```
public class CustomWebApplicationFactory
    : WebApplicationFactory<Startup>
{
    protected override void ConfigureWebHost(
        IWebHostBuilder builder)
    {
        builder.ConfigureTestServices(services =>
        {
            services.RemoveAll<ICurrencyConverter>();
            services.AddSingleton
                <ICurrencyConverter, StubCurrencyConverter>();
        });
    }
}
```

Derive from WebApplicationFactory.

Add custom configuration for your application.

There are many functions available to override. This is equivalent to calling WithWebHostBuilder.

In this example, we override `ConfigureWebHost` and configure the test services for the factory.[7] You can use your custom factory in any test by injecting it as an `IClassFixture`, as you have before. For example, the following listing shows how you would update listing 23.15 to use the custom factory defined in listing 23.16.

Listing 23.17 Using a custom `WebApplicationFactory` in an integration test

```
public class IntegrationTests:
    IClassFixture<CustomWebApplicationFactory>
{
    private readonly CustomWebApplicationFactory _fixture;
    public IntegrationTests(CustomWebApplicationFactory fixture)
    {
        _fixture = fixture;
    }

    [Fact]
    public async Task ConvertReturnsExpectedValue()
    {
        HttpClient client = _fixture.CreateClient();

        var response = await client.GetAsync("/api/currency");

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();

        Assert.Equal("3", content);
    }
}
```

Implement the IClassFixture interface for the custom factory.

Inject an instance of the factory in the constructor.

The client already contains the test service configuration.

The result confirms the test service was used.

[7] `WebApplicationFactory` has many other methods you could override for other scenarios. For details, see http://mng.bz/mgq8.

You can also combine your custom `WebApplicationFactory`, which substitutes services that you always want to replace, with the `WithWebHostBuilder()` method to override additional services on a per-test basis. That combination gives you the best of both worlds: reduced duplication with the custom factory, and control with the per-test configuration.

Running integration tests using your real app's configuration provides about the closest you'll get to a guarantee that your app is working correctly. The sticking point in that guarantee is nearly always external dependencies, such as third-party APIs and databases.

In the final section of this chapter, we'll look at how to use the SQLite provider for EF Core with an in-memory database. You can use this approach to write tests for services that use an EF Core database context, without needing access to a real database.

23.6 Isolating the database with an in-memory EF Core provider

In this section you'll learn how to write unit tests for code that relies on an EF Core `DbContext`. You'll learn how to create an in-memory database, and the difference between the EF in-memory provider and the SQLite in-memory provider. Finally, you'll see how to use the in-memory SQLite provider to create fast, isolated tests for code that relies on a `DbContext`.

As you saw in chapter 12, EF Core is an ORM that is used primarily with relational databases. In this section I'm going to discuss one way to test services that depend on an EF Core `DbContext` without having to configure or interact with a real database.

> **NOTE** To learn more about testing your EF Core code, see *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021), http://mng.bz/5j87.

The following listing shows a highly stripped-down version of the `RecipeService` you created in chapter 12 for the recipe app. It shows a single method to fetch the details of a recipe using an injected EF Core `DbContext`.

Listing 23.18 `RecipeService` to test, which uses EF Core to store and load entities

```
public class RecipeService
{
    readonly AppDbContext _context;
    public RecipeService(AppDbContext context)
    {
        _context = context;
    }
    public RecipeViewModel GetRecipe(int id)
    {
        return _context.Recipes
            .Where(x => x.RecipeId == id)
            .Select(x => new RecipeViewModel
```

An EF Core DbContext is injected in the constructor.

Uses the DbSet<Recipes> property to load recipes and creates a RecipeViewModel

```
            {
                Id = x.RecipeId,
                Name = x.Name
            })
            .SingleOrDefault();
    }
}
```

Writing unit tests for this class is a bit of a problem. Unit tests should be fast, repeatable, and isolated from other dependencies, but you have a dependency on your app's `DbContext`. You probably don't want to be writing to a real database in unit tests, as it would make the tests slow, potentially unrepeatable, and highly dependent on the configuration of the database: a fail on all three requirements!

> **NOTE** Depending on your development environment, you *may* want to use a real database for your integration tests, despite these drawbacks. Using a database like the one you'll use in production increases the likelihood you'll detect any problems in your tests. You can find an example of using Docker to achieve this in Microsoft's "Testing ASP.NET Core services and web apps" documentation: http://mng.bz/zxDw.

Luckily, Microsoft ships two *in-memory* database providers for this scenario. Recall from chapter 12 that when you configure your app's `DbContext` in `Startup.ConfigureServices()`, you configure a specific database provider, such as SQL Server:

```
services.AddDbContext<AppDbContext>(options =>
   options.UseSqlServer(connectionString);
```

The in-memory database providers are alternative providers designed *only for testing*. Microsoft includes two in-memory providers in ASP.NET Core:

- *Microsoft.EntityFrameworkCore.InMemory*—This provider doesn't simulate a database. Instead, it stores objects directly in memory. It isn't a relational database as such, so it doesn't have all the features of a normal database. You can't execute SQL against it directly, and it won't enforce constraints, but it's fast.
- *Microsoft.EntityFrameworkCore.Sqlite*—SQLite is a relational database. It's very limited in features compared to a database like SQL Server, but it's a true relational database, unlike the in-memory database provider. Normally a SQLite database is written to a file, but the provider includes an in-memory mode, in which the database stays in memory. This makes it much faster and easier to create and use for testing.

Instead of storing data in a database on disk, both of these providers store data in memory, as shown in figure 23.9. This makes them fast and easy to create and tear down, which allows you to create a new database for every test to ensure your tests stay isolated from one another.

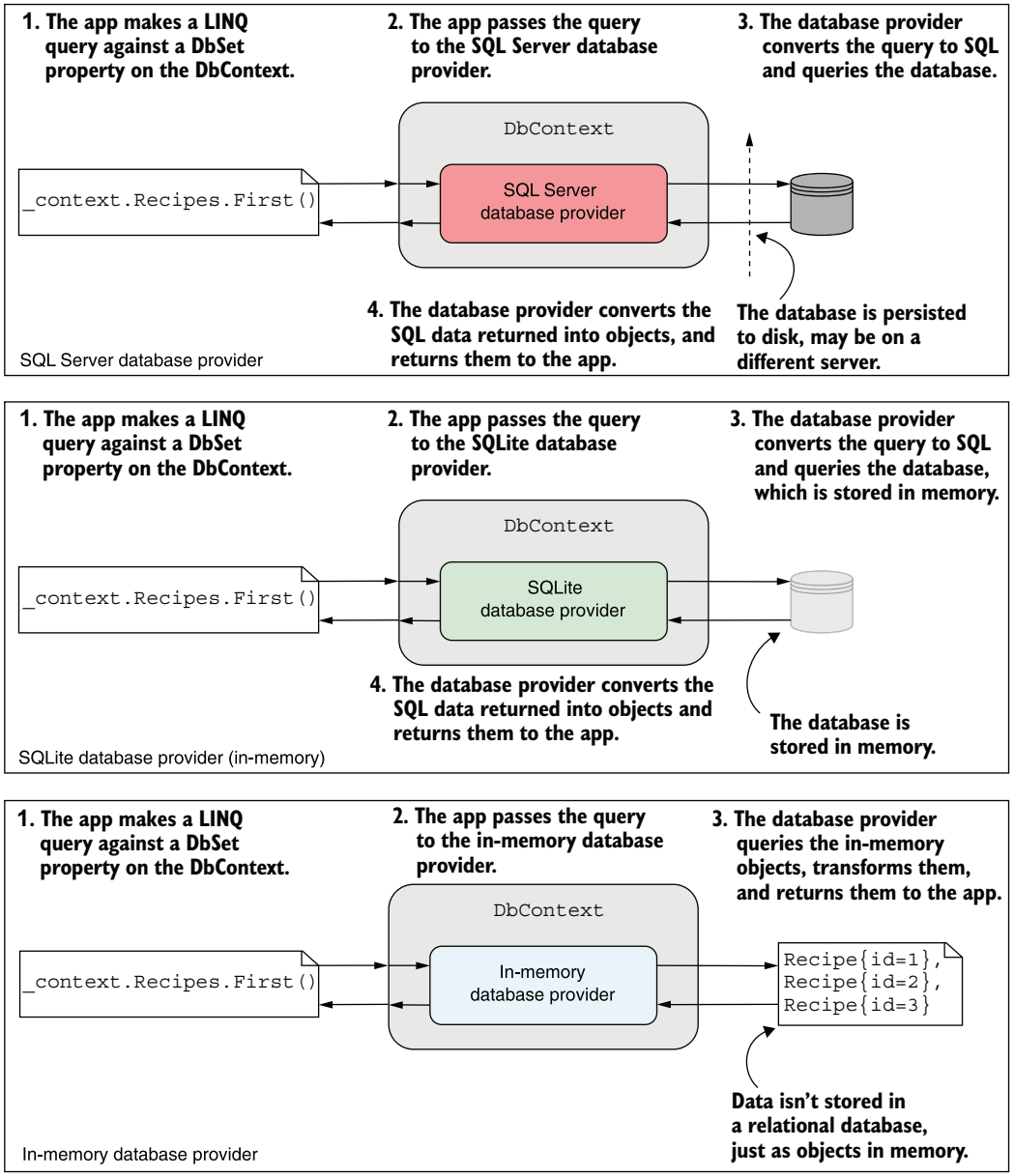


Figure 23.9 The in-memory database provider and SQLite provider (in-memory mode) compared to the SQL Server database provider. The in-memory database provider doesn't simulate a database as such. Instead, it stores objects in memory and executes LINQ queries against them directly.

> **NOTE** In this section, I describe how to use the SQLite provider as an in-memory database, as it's more full-featured than the in-memory provider. For details on using the in-memory provider, see Microsoft's "EF Core In-Memory Database Provider" documentation: http://mng.bz/hdIq.

To use the SQLite provider in memory, add the Microsoft.EntityFrameworkCore.Sqlite package to your test project's .csproj file. This adds the `UseSqlite()` extension method, which you'll use to configure the database provider for your unit tests.

Listing 23.19 shows how you could use the in-memory SQLite provider to test the `GetRecipe()` method of `RecipeService`. Start by creating a `SqliteConnection` object and using the `"DataSource=:memory:"` connection string. This tells the provider to store the database in memory and then open the connection.

> **WARNING** The in-memory database is destroyed when the connection is closed. If you don't open the connection yourself, EF Core will close the connection to the in-memory database when you dispose of the `DbContext`. If you want to share an in-memory database between `DbContexts`, you must explicitly open the connection yourself.

Next, pass the `SqlLiteConnection` instance into the `DbContextOptionsBuilder<>` and call `UseSqlite()`. This configures the resulting `DbContextOptions<>` object with the necessary services for the SQLite provider and provides the connection to the in-memory database. By passing this options object into an instance of `AppDbContext`, all calls to the `DbContext` result in calls to the in-memory database provider.

Listing 23.19 Using the in-memory database provider to test an EF Core `DbContext`

Opens the connection so EF Core won't close it automatically

Configures an in-memory SQLite connection using the special "in-memory" connection string

```
[Fact]
public void GetRecipeDetails_CanLoadFromContext()
{
    var connection = new SqliteConnection("DataSource=:memory:");
    connection.Open();

    var options = new DbContextOptionsBuilder<AppDbContext>()
        .UseSqlite(connection)
        .Options;

    using (var context = new AppDbContext(options))
    {
        context.Database.EnsureCreated();
        context.Recipes.AddRange(
            new Recipe { RecipeId = 1, Name = "Recipe1" },
            new Recipe { RecipeId = 2, Name = "Recipe2" },
            new Recipe { RecipeId = 3, Name = "Recipe3" });
        context.SaveChanges();
    }
```

Creates an instance of DbContext-Options<> and configures it to use the SQLite connection

Creates a DbContext and passes in the options

Ensures the in-memory database matches EF Core's model (similar to running migrations)

Adds some recipes to the DbContext

Saves the changes to the in-memory database

This example follows the standard format for any time you need to test a class that depends on an EF Core `DbContext`:

1. Create a `SqliteConnection` with the `"DataSource=:memory:"` connection string and open the connection.
2. Create a `DbContextOptionsBuilder<>` and call `UseSqlite()`, passing in the open connection.
3. Retrieve the `DbContextOptions` object from the `Options` property.
4. Pass the options to an instance of your `DbContext` and ensure the database matches EF Core's model by calling `context.Database.EnsureCreated()`. This is similar to running migrations on your database, but it should *only* be used on test databases. Create and add any required test data to the in-memory database and call `SaveChanges()` to persist the data.
5. Create a new instance of your `DbContext` and inject it into your test class. All queries will be executed against the in-memory database.

By using two separate `DbContexts`, you can avoid bugs in your tests due to EF Core caching data without writing it to the database. With this approach, you can be sure that any data read in the second `DbContext` was persisted to the underlying in-memory database provider.

This was a very brief introduction to using the SQLite provider as an in-memory database provider, and EF Core testing in general, but if you follow the setup shown in listing 23.19, it should take you a long way. The source code for this chapter shows how you can combine this code with a custom `WebApplicationFactory` to use an in-memory database for your integration tests. For more details on testing EF Core, including additional options and strategies, see *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021).

Summary

- Unit test apps are console apps that have a dependency on the .NET Test SDK, a test framework such as xUnit, MSTest, or NUnit, and a test runner adapter. You can run the tests in a test project by calling `dotnet test` from the command line in your test project or by using the Test Explorer in Visual Studio.

- Many testing frameworks are compatible with the .NET Test SDK, but xUnit has emerged as an almost *de facto* standard for ASP.NET Core projects. The ASP.NET Core team themselves use it to test the framework.
- To create an xUnit test project, choose xUnit Test Project (.NET Core) in Visual Studio or use the `dotnet new xunit` CLI command. This creates a test project containing the Microsoft.NET.Test.Sdk, xunit, and xunit.runner.visualstudio NuGet packages.
- xUnit includes two different attributes to identify test methods. `[Fact]` methods should be public and parameterless. `[Theory]` methods can contain parameters, so they can be used to run a similar test repeatedly with different parameters. You can provide the data for each `[Theory]` run using the `[InlineData]` attribute.
- Use assertions in your test methods to verify that the system under test (SUT) returned an expected value. Assertions exist for most common scenarios, including verifying that a method call raised an exception of a specific type. If your code raises an unhandled exception, the test will fail.
- Use the `DefaultHttpContext` class to unit test your custom middleware components. If you need access to the response body, you must replace the default `Stream.Null` with a `MemoryStream` instance and manually read the stream after invoking the middleware.
- API controllers and Razor Page models can be unit tested just like other classes, but they should generally contain little business logic, so it may not be worth the effort. For example, the API controller is tested independently of routing, model validation, and filters, so you can't easily test logic that depends on any of these aspects.
- Integration tests allow you to test multiple components of your app at once, typically within the context of the ASP.NET Core framework itself. The Microsoft.AspNetCore.TestHost package provides a `TestServer` object that you can use to create a simple web host for testing. This creates an in-memory server that you can make requests to and receive responses from. You can use the `TestServer` directly when you wish to create integration tests for custom components like middleware.
- For more extensive integration tests of a real application, you should use the `WebApplicationFactory` class in the Microsoft.AspNetCore.Mvc.Testing package. Implement `IClassFixture<WebApplicationFactory<Startup>>` on your test class and inject an instance of `WebApplicationFactory<Startup>` into the constructor. This creates an in-memory version of your whole app, using the same configuration, DI services, and middleware pipeline. You can send in-memory requests to your app to get the best idea of how your application will behave in production.
- To customize the `WebApplicationFactory`, call `WithWebHostBuilder()` and call `ConfigureTestServices()`. This method is invoked after your app's standard

DI configuration. This enables you to add or remove the default services for your app, such as to replace a class that contacts a third-party API with a stub implementation.

- If you find you need to customize the services for every test, you can create a custom `WebApplicationFactory` by deriving from it and overriding the `ConfigureWebHost` method. You can place all your configuration in the custom factory and implement `IClassFixture<CustomWebApplicationFactory>` in your test classes, instead of calling `WithWebHostBuilder()` in every test method.
- You can use the EF Core SQLite provider as an in-memory database to test code that depends on an EF Core database context. You configure the in-memory provider by creating a `SqliteConnection` with a `"DataSource=:memory:"` connection string. Create a `DbContextOptionsBuilder<>` object and call `UseSqlite()`, passing in the connection. Finally, pass `DbContextOptions<>` into an instance of your app's `DbContext` and call `context.Database.EnsureCreated()` to prepare the in-memory database for use with EF Core.
- The SQLite in-memory database is maintained as long as there's an open `SqliteConnection`. By opening the connection manually, the database can be used with multiple `DbContexts`. If you don't call `Open()` on the connection, EF Core will close the connection (and delete the in-memory database) when the `DbContext` is disposed of.